Towards Structured State Threading in Prolog

Dragan Ivanović[1], José F. Morales[2], Manuel Carro[1], and Manuel Hermenegildo[1,3]

[1] School of Computer Science, T. University of Madrid (UPM)
(idragan@clip.dia.fi.upm.es, {mcarro, herme}@fi.upm.es)
[2] Universidad Complutense de Madrid (UCM) (jfmc@fdi.ucm.es)
[3] IMDEA Software, Spain

Abstract. It is very often the case that programs require passing, maintaining, and updating some notion of state. Prolog programs often implement such stateful computations by carrying this state in predicate arguments (or, alternatively, in the internal database). This often causes code obfuscation, complicates code reuse, introduces dependencies on the data model, and is prone to incorrect propagation of the state information among predicate calls. To partly solve these problems, we introduce *contexts* as a consistent mechanism for specifying implicit arguments and its threading in clause goals. We propose a notation and an interpretation for contexts, ranging from single goals to complete programs, give an intuitive semantics, and describe a translation into standard Prolog. We also discuss a particular light-weight implementation in Ciao Prolog, and we show the usefulness of our proposals on a series of examples and applications, including code directly using contexts, DCGs, extended DCGs, logical loops and other custom control structures.

Keywords: Stateful Computation, State Abstraction, Contexts, Prolog Extensions.

1 Introduction

Programming stateful computations in Prolog often requires the state information to be passed as arguments to predicates, which complicates source code and increases chances for programming errors. The alternative of storing the state in a dynamic database makes code much more difficult to reason about, both for humans and for automatic tools, since the database behavior is at odds with the referential transparency of Logic Programming. Besides, asserting and retracting bring about an important performance overhead.

This paper presents context propagation and scoping as a new approach to help produce code which is easier to write and to maintain in this situation. The approach we present extends the Prolog syntax with what we call *contextual notation*.[4] It includes state definitions, context expansions, and scoped contexts, and tries to go beyond what other, related proposals, such as EDCGs [2], Mercury state variables [3], global variables [4], or mutable variables [5] can directly express (See Section 3 for a more in-depth discussion). The semantics of a Prolog program using this contextual notation is defined in terms of context propagation in control structures and clauses.

[4] Not to be confused with contextual logic programming [1] (see Section 4.3).

```
calc_new([]).

calc_top(X, [X|_]).

calc_add([X, Y|L], [Z|L]):-
    Z is X+Y.

% More operations ...
```

(a) First version of a calculator.

```
calc_new(c([], 0)).

calc_top(X, c([X|_], _)).

calc_add(c([X,Y|L], R), c([Z|L], R)):-
    Z is X+Y.

calc_sto(c([X|L], _), c([X|L], X)).

% More operations ...
```

(b) Improved version with a register.

Fig. 1. Two standard Prolog implementations of a stack-based calculator

Throughout the paper we will use a simple example to illustrate some of the problems which appear when writing and maintaining Prolog programs which need to transform states and compute with them. Figure 1(a) shows a Prolog implementation of part of a stack-based calculator, where the stack is modeled as a list. Predicate `calc_new/1` creates a new empty calculator, `calc_top/2` gets the top of the stack, and `calc_add/2` performs addition by removing the two topmost stack elements and leaving their addition on the top of the stack again. We can of course expect of a calculator to implement more operations, such as subtraction, division, roots, etc. These calculator operations are probably meant to be used from a loop such as

```
top_level(State):-
    read_command(Command),
    execute(Command, State, NewState),
    top_level(NewState).
```

where the transition from `State` to `NewState` is performed by the aforementioned predicates, one for every `Command`.

Were we to improve the calculator by introducing a memory register, we would need to update our model to, e.g., a structure `c/2`, where the first argument models the stack and the second one models the memory register (Figure 1(b)).

A new predicate `calc_sto/2` copies the top of the stack onto the register, and the rest of the operations have to be adapted, regardless of whether they use the new feature or not. Of course, incorrectly recoding the necessary operations can cause hard-to-catch bugs which will lead to incorrect results and even to program failure. This is a classical example of poor code reusability stemming from a *dependency on the state model*. Type checking can help in alleviating this problem, but even in this case the issue of actually updating the affected code remains.

Additionally, in `calc_add/2` we had to manually replicate the stack from the input to the output states and to explicitly thread the value of the rest of stack `L` and the register `R`. The need to correctly *manually thread state components* is of course an additional source of programming errors, more even so if there are several states which have to be threaded at the same time.

In the rest of the paper we will present our proposal for contextual notation, describe how it is compiled into plain Prolog (which will help us in giving it a more

```
:- def_state calc := stack.

calc_new / -calc:-
    set([]) / stack.


calc_top(X) / +calc :-
    get([X|_]) / stack.

calc_add / calc:-
    get([X,Y|L]) / stack,
    Z is X+Y,
    set([Z|L]) / stack.
```

(a) Contextual version of a calculator.

```
:- def_state calc := ^c(stack, reg).

calc_new / -calc:-
    set([]) / stack,
    set(0) / reg.

calc_top(X) / +calc :-
    get([X|_]) / stack.

calc_add / calc:-
    get([X,Y|L]) / stack,
    Z is X+Y,
    set([Z|L]) / stack.

calc_sto / calc:-
    get([X|_]) / stack,
    set(X) / reg.
```

(b) Calculator with a register.

Fig. 2. Contextual notation version of the code snippets in Figure 1

formal semantics), and describe some particularly interesting extensions. We finally present some application examples and discuss related work.

2 Contextual Notation: Syntax and Semantics

We will present now the syntax of our contextual notation and we will give an intuitive description of their semantics. We will introduce also a sketch of a translation scheme into Prolog, which besides showing how contextual notation introduces just a negligible (and always constant) overhead, will give additional insight into the meaning and capabilities of contextual programs.

2.1 Contextual Notation: an Example

Rather than specifying formally the syntax and semantics of contexts, we will show a simple example and use it to give an intuition of what our notation achieves. In a nutshell, a context is a description of implicit arguments that represent input and output (structured) state information handled by a predicate. By input we mean the state at call time, and by output we mean the updated state upon successful execution of a predicate.

Figure 2 shows contextual version of the previously shown calculator examples. The leftmost code defines a `calculator` as having a single component, the `stack`. Note that we do not define anything else regarding the stack: it is just the name of a component. However, just by assigning a name to that component, we are reducing the possibility to assign something which is not a stack to a stack by a mistake.

Every predicate is marked as receiving a `calc` context (in the construction `pred_name / calc`). Inside the predicate definition, the components of the `calculator` are visible. A context starting with - (minus sign) marks the predicate

as being output only: that is the case of `calc_new/1`, which `set`s the value associated to the `stack` component of the context to `[]`.[5] The + and the - prefixes used in contexts should not be confused with the usual notation for modes in Prolog.

Contexts starting with a + (plus sign) are input only. This is the case of `calc_display/1`, which outputs in the screen the element at the stack top. The associated state remains unchanged. Finally, contexts without sign (as in `calc_add/1`) are input / output: their values can be consulted, as done to access the two topmost elements in the stack, and set, as done to store the result of the addition. Note that `get` is using unification with logical variables at the same time it retrieves the value associated to a context. `stack` appears at most once in every predicate call, while clearly accessing its value and setting a new one needs two variables (similarly for the calculator). Ensuring proper threading is implicit and automatically performed by the translation into Prolog. This maybe becomes clearer in the contextual version of the toplevel loop of the calculator

```
top_level / +calc :-
    read_command(Command),
    execute(Command) / calc,
    top_level / +calc.
```

where every `execute` receives a calculator state and returns a new calculator state.

A distinguishing feature of the contextual notation is that, unlike EDCGs or the state variables of Mercury, it is able to directly represent structured state. Figure 2, right, shows the calculator with a register written using contexts. Note that the `calculator` definition has now two components instead of one, which can be accessed inside a predicate definition whose context is a `calc`. The translation of contexts adds the corresponding structures as arguments in clause heads, and threads their components across clause bodies. With argument structures directly represented in clause heads, the translated code makes use of clause-head matching and indexing facilities of the underlying engine, instead of relying on runtime checking and intermediate packing/unpacking of state arguments.[6]

The initialization operation has to change in order to give a proper value to the register,[7] and a new operation to store in this register is added, as before. However, the code for the other two operations remains unchanged, as the contextual notation will make sure that `calculator` components which are not `set` are not changed, and the components are accessed by name, so that their position in a structure is irrelevant. In absence of this capability, manual deconstruction and construction of structures would be needed, with the increased possibility of mistakes.

The combination of threading with structured state naming is able to give solutions to the two problems identified before: independence of the state model when possible, and automatic state threading, while keeping a reasonable declarative reading which features backtracking and logical variables.

[5] Context definitions work with `get` and `set` operations with the obvious meaning, in a way similar to VDM. Operations with nicer, more compact syntax, can be defined, but for the sake of clarity we prefer to stick to `get` and `set` in these examples.

[6] For illustration of making use of different clauses for different structured states, see the Example 1 on page 8.

[7] However, note that if the initial value of the register were not defined, no changes to `calc_new/1` would have been needed.

$s ::= n$	(named state definition)	$c ::= s$	(input-output state)
$\mid x$	(state component)	$\mid + s$	(input-only state)
$\mid \hat{}\, a$	(atomic term)	$\mid - s$	(output-only state)
$\mid \hat{}\, v$	(logical variable)	$\mid *$	(empty context; no state)
$\mid \hat{}\, f(s,s,\ldots,s)$	(structure, arity > 0)	$\mid c, c$	(composition of sub-contexts)

Table 1. Syntax for states.

Table 2. Syntax for contexts.

2.2 States, Components, and Contexts

We will now introduce the notation that allows us to specify the structure of state information and to name its components. State definitions have the form:

```
:- def_state n := s .
```

where n is an atom that is the name of the state, and s is the actual state specification.

Definition 1 (State Specification). *A state specification s is constructed by a finite number of applications of the recursive rule in Table 1.*

Use of named state definitions in state specifications makes it possible to reuse previous state specifications when writing new ones. Note that since state specifications have the *shape* of a Prolog term, when a Prolog term has to be interpreted as such, and not as a state name, it is escaped using a prefixed caret. See, for example, the definition of the `calc` structure in Figure 2, right.

The notation for a goal G in a context c is G/c. The syntax for G is that of regular Prolog goals, and the syntax for contexts is shown in Table 2. Figure 2 has examples for contexts starting with `-`, `+`, and without prefix. Our translation into Prolog will append extra arguments to G, according to the type of context which is applied to G, and thread these arguments in clause bodies according to the state names and whether they are input or output. The *empty* context is introduced in order to uniformly view and deal with predicates without context in an environment where it is needed. Context composition makes it possible to pack several contexts in a single one and write predicates which act on several (structured) states at the same time.

2.3 The Translation for States and Contexts

State and context specifications are translated into additional predicate arguments by means of their *interpretation*. An interpretation builds terms in which state components are replaced with logical variables that serve as value place-holders. Thus, the difference between input and output interpretations is in the variables used for that purpose. Table 3 shows the translation of goals in context for two different context definitions — those used in the calculator examples.

The interpretation of components at every point in a clause is given by a mapping from component names to variables which whill be used to assign logical variables to hold the component value at every program point.

Definition 2 (Component Binding). *A* component binding θ *is an injective function that maps every state component x to a distinct logical variable $\theta(x)$ (unused in code).*

Goal / Context	Calculator 1	Calculator 2
`calc_new / -calc`	`calc_new(StackOut)`	`calc_new(c(StackOut, RegOut))`
`calc_top(X) / +calc`	`calc_top(StackIn, StackIn)`	`calc_top(StackIn, c(StackIn, RegIn))`
`calc_add / calc`	`calc_add(StackIn, StackOut)`	`calc_add(c(StackIn, RegIn), c(StackOut, RegOut))`
`get([X,Y|L]) / stack`	`[X,Y|L]=StackIn`	`[X,Y|L]=StackIn`
`set(0) / reg`	—	`0=RegOut`

Table 3. Predicates in contexts and their translation for two context definitions.

Before and after a predicate in context is called, state components are mapped to input and output logical variables (when needed: not always both mappings are necessary). These two mappings are stored in an *input binding* (before a state update) and an *output binding* (after a state update), and are used to generate input and interpretations for states and for contexts as follows:

Definition 3 (State and Context Interpretations). *(1) Given a state s, an input binding θ^i, and an output binding θ^o, the input and the output interpretations s^i and s^o are terms defined as:*

$$(x)^i = \theta^i(x) \qquad (x)^o = \theta^o(x)$$
$$(\hat{}a)^i = a \qquad (\hat{}a)^o = a$$
$$(\hat{}v)^i = v \qquad (\hat{}v)^o = v$$
$$(\hat{}f(s_1,\ldots,s_n))^i = f(s_1^i,\ldots,s_n^i) \qquad (\hat{}f(s_1,\ldots,s_n))^o = f(s_1^o,\ldots,s_n^o)$$
$$(n)^i = s^i_{\text{def}} \qquad (n)^o = s^o_{\text{def}}$$

where s_{def} is a state definition for n introduced with a `def_state` *declaration.*

(2) Given a context c defined as in Table 2, an input binding θ^i, and an output binding θ^o, the context interpretation $i(c)$ is a sequence of additional argument terms defined as:

$$i(s) = \langle s^i, s^o \rangle$$
$$i(+s) = \langle s^i \rangle \qquad i(-s) = \langle s^o \rangle$$
$$i(*) = \Lambda \qquad i(c_1, c_2) = i(c_1)||i(c_2)$$

where $||$ stands for sequence concatenation, and Λ is the empty sequence.

State and context interpretations are the key for translating constructs of the form G/c to standard Prolog goals. Given a G/c construct, and an input interpretation θ^i, we wish to formulate a rule for obtaining a translation of G/c and a new, resulting binding θ^o. For the latter, we wish to ensure that mutable state components receive output bindings that are different from the input bindings. If a state component does not appear in an output of c, we consider it immutable; otherwise, the output value of a component can be different from its input value, depending on the target predicate. We formalize the notion of mutability in the following way.

Definition 4 (Component Mutability). *A state component x* is mutable in context c *(we write $\mu_c(x)$) if and only if it appears in some sub-context s or $-s$ of c.*

```
calc_new(c(Stack, Reg)):-
    []=Stack, 0=Reg.

calc_top(X, c(Stack, Reg)) :-
    [X|_]=Stack.
```

```
calc_add(c(S0, R0), c(S1, R0)):-
    [X,Y|L]=S0,
    Z is X+Y,
    [Z|L]=S1.

calc_sto(c(S0, R0), c(S0, R1)):-
    [X|_]=S0, X=R1.
```

Fig. 3. A goal-by-goal translation.

Definition 5 (Atomic Goal Expansion). *Given a goal $G = g(\bar{t})$, where $\bar{t}$ is a (possibly empty) argument list, a context c, and an input binding θ^i, the* expansion of G/c *is defined as:*

$$G[\![c]\!] = g(\bar{t}||i(c)), \tag{1}$$

with the output binding:

$$\theta^o(x) = \begin{cases} \theta^i(x), & \neg\mu_c(x) \\ v, & \mu_c(x) \end{cases} \tag{2}$$

where v is a fresh (i.e. previously unused) variable.

As an example, in Table 3, the context in `calc_add / calc` is translated into two different arguments containing structures. The sequence of arguments has been subject to deforestation to give two different arguments.

According to the above definition, $=(T)/+s$ would converted into $T = s^i$ and would unify T with the input state of s; similarly with $=(T)/-s$, thereby allowing to *get* or *set* the state of s. For the sake of clarity, we introduce the special notational shorthand `get`$(T)/s$ and `set`$(T)/s$.

Besides, it can be easily verified that expansion of $G/*$ produces $G[\![*]\!] = G$ with $\theta^o = \theta^i$, because $i(*) = \Lambda$ and for all components x we have $\neg\mu_*(x)$. This allows us to treat all goals in a clause (except control structures) in the subsequent analysis as G/c constructs, even in trivial cases where $c = *$.

2.4 Threading States

Ensuring that context components are correctly threaded across control structures and in clauses needs to take into account alignment of component bindings both before and after expansions of each G/c. Figure 3 portrays a translation into Prolog of the code in Figure 1, left, where bindings are chained to ensure proper threading. We set out the basic translation criteria for ensuring correct threading as follows.

Definition 6 (Sequence Translation). *A sequence $\sigma_n = G_1/c_1, \ldots, G_n/c_n$ $(n \geq 1)$, given input state θ_0, is translated into $\Sigma_n = G_1[\![c_1]\!], \ldots, G_n[\![c_n]\!]$ so that:*

(i) the input binding for $G_1[\![c_1]\!]$ is θ_0;
(ii) the output binding θ_i of $G_i[\![c_i]\!]$ $(1 \leq i < n)$ coincides with the input binding for expanding G_{i+1}/c_{i+1}; and
(iii) the output binding of the entire sequence is θ_n.

The sequence translation rule can be applied straightforwardly, by sequentially expanding goals using the output binding obtained from an expansion as an input binding for the next expansion. The following table shows an example of sequence translation with state threading in the body of `calc_add` (Figure 2).

```
:- def_state node :=
    ^t( left, elem, right).
:- def_state nil  := ^[].

empty / -nil.

check(X) / +node :-
    (  get(X) / elem
    ;  check(X) / +left
    ;  check(X) / +right
    ).
```

```
ins(X) / node :-
    get(E) / elem,
    (  X @< E -> ins(X) / left
    ;  X @> E -> ins(X) / right
    ;  X = E
    ).

ins(X) / (+nil, -node) :-
    set(X) / elem,
    empty / -left,
    empty / -right.
```

```
traverse / (+tree, +dl) :-
    traverse1 / (+tree, dl),
    set([]) / dl.

traverse1 / (+nil,  dl).
traverse1 / (+node, dl):-
    traverse1 / (+left, dl),
    get(E) / elem,
    get([E|L]) / dl,
    set(L) / dl,
    traverse1 / (+right, dl).
```

Fig. 4. A tree handling example

k	θ_{k-1}(stack)	θ_{k-1}(reg)	G_k/c_k	μ_{c_k}(stack)?	μ_{c_k}(reg)?	$G_k[\![c_k]\!]$
1	S0	R0	`get([X,Y|L]) / stack,`	No	No	`[X,Y|L]=S0,`
2	S0	R0	`(Z is X+Y) / *,`	No	No	`Z is X+Y,`
3	S0	R0	`set([Z|L]) / stack.`	Yes	No	`[Z|L]=S1.`
4	S1	R0	← the resulting binding from sequence threading			

Finally, we define the criteria for correct threading of states between the head and the body of a clause.

Definition 7 (Clause Translation). *A clause $G/c \leftarrow \sigma_n$ is translated into $G[\![c]\!] \leftarrow \Sigma_n$ so that:*

(i) the input binding θ_0 is the same for both the head expansion $G[\![c]\!]$ and Σ_n;
(ii) the output binding θ^o_G of $G[\![c]\!]$ is the same as the output binding θ_n from Σ_n.

In application of the clause translation rule we note that the expansion of G/c introduces fresh variables in the output binding θ^o_G for all components that are mutable in c. Therefore, to ensure fulfillment of the requirement (ii), we substitute output bindings in θ^o_G with the output bindings from θ_n. To continue the same example, expansion of the head calc_add/calc from Fig. 2(b) produces:

θ(stack)	θ(reg)	$h(\bar{t})/c$	μ_c(stack)?	μ_c(reg)?	$h'(\bar{t})$
S0	R0	`calc_add / calc`	Yes	Yes	`calc(c(S0,R0), c(S2,R1))`
S2	R1	← the resulting binding from head expansion			

where S2 and R1 are fresh variables. To meet condition (ii), we substitute S2 with S1 and R1 with R0, and finally obtain the translation of calc_add on Fig. 3.

Example 1. The code on Fig. 4 shows some usual operations on ordered binary trees. We have two state definition: `nil` that denotes the atom `[]` (representing the empty tree), and `node` that has three components – the element at the node `elem`, and the left and the right subtrees `left` and `right`, respectively.

The predicate `empty/1` outputs an empty tree. The predicate `check/2` checks for occurrence of a term in an input tree. Since by Definition 5 the argument list is expanded with an input interpretation of `node`, the head of the `check` clause is not matched if the input is a `nil`. The predicate `ins/3` inserts its first argument into a tree represented by the context. The first clause of `ins/3` operates on a `node`, while the second clause takes a `nil` for an input, and produces a `node` as an output. Note

that, by Definition 3(2), contexts `node` and `(+node,-node)` have the same interpretation. Thus, a clause is selected depending on the actual call-time argument.

The predicate `traverse/2` shows how we can handle difference lists in much the same way we do with DCGs. We accept the input (i.e. the prefix) of the difference list `dl`, use the auxiliary predicate `traverse1/3`, and finally set of the output (i.e. the suffix) of `dl` to `[]`.

Note that we can easily extend these operations to other ordered collections by defining distinct structured states and adding the corresponding (discontinuous or multi-file) clauses for `check/2`, `ins/3` and `traverse1/3`.

2.5 Implementation of the Translation

The work in this paper has been implemented using Ciao packages, which make it possible to write compiler *plug-ins* which extend syntax and provide additional facilities. A benefit of this translation-based approach, in addition to the portability across other Prolog systems, is the availability of other language extensions, different evaluation techniques (such as tabling),program analysis, verification, and run-time checks [6, 7], etc. The process of translating clauses containing context notation to standard Prolog clauses is implemented as a code transformation at compile time. State components are included in threading upon first encounter in a context, which may be in the head of the clause of anywhere in the body. The use of an input binding of a state component that has not been previously encountered can be thus detected, and a compiler warning on the use of previously uninitialized state issued.

2.6 Applicability

The contextual extensions described in this work have been extensively used in the Ciao system for several months, in order to rewrite some parts and develop new code especially for the documentation generator LPdoc [8]. Its has been instrumental in reducing the complexity, managing, maintaining, and redesigning large pieces of Prolog code. We experienced no performance degradation during the port, as the translation and the context language are precise enough not to introduce any overhead, and it was conveniently used in the right places.

3 Extensions to the Basic Framework

There are some interesting features which, for conciseness, we did not include in the previous description, but which are interesting as they provide the programmer with tools which help in making the code easier to write and to understand. We will here briefly describe some of them.

3.1 Towards Type-Checking of States

As noted in Section 1, being able to ensure type-conformance, for some notion of types, increases confidence on program correctness by helping to flag bugs which, in our case, can stem from, for example, a wrong assignment. The current implementation of context variables includes the possibility of specifying the type (e.g., integer, atom, structure with a given name and arity) that some components of a state must have in order for the state to be validated, and also automatically generate type-checking predicates which are called automatically at run-time when such

a value is changed, and which throw an exception if type annotations are violated. We note that this can be currently used even in host systems which do not have infrastructure for type inference or checking.

However, we plan to fully integrate this capability with the static type inference and run-time type checking of Ciao [9, 10]. This will allow us to reutilize the machinery already developed, and be able to to decide at compile time type correctness when possible, and generate runtime checks otherwise, in a way which is homogeneous across the code supported by the system — e.g., both for contextual and non-contextual code, and in context and non-context positions.

3.2 State Navigation

It is often useful to be able to access sub-components of a component that is structured itself. To enable that, we extend the syntax for states (Table 1) with component typing of the form $x :: n$, to denote that the component x is itself structured according to the globally named state definition n. We further extend the syntax of state specification with the dot notation $x.y$, referring to the sub-component y of the component x. In fact, we introduce a generalization of an untyped component name x into a component path $\bar{x} = x_1.x_2.\cdots.x_n$ $(n \geq 1)$.

Introduction of component paths changes the threading and translation rules in the sense that after the first appearance of $\bar{x}.y$ in a context, all sub-components of $\bar{x}$ of the form $\bar{x}.\alpha$ start to be threaded, and always substitute $\theta^i(\bar{x})$ with a structure built of $\theta^i(\bar{x}.\alpha)$s after the definition of n. Also, from that point on, $\theta^o(\bar{x})$ is always the corresponding structure composed of $\theta^o(\bar{x}.\alpha)$s.

3.3 Syntactic Sugar for Setters and Getters

Shorthands for the `set` and `get` operations are available in order to make code more compact and with a look more similar to that of imperative programming:

$s<-v \equiv$ `set(`v`)/`s	Give contextual variable s value v
`T=`s`@` $\equiv$ `get(T)/`s	Obtain in t the value of contextual variable s

The (`<-`) notation acts as if it were a destructive assignment on a given contextual variable or component, while x`@`, in a term position, is replaced by $\theta^i(x)$ for the θ^i currently active at that position.

3.4 Implicit Context

Each clause in a predicate with a context requires the contextual arguments to be written explicitly. The same happens for contexts which are shared among several predicates (e.g., when writing an ADT and encapsulating its operations): declaring that shared context would spare the programmer from writing it explicitly. In Figure 1, the calculator example (Figure 5) is rewritten with a more concise syntax, where the clauses in the common context `calc` and the state definition are grouped together. The `:- var` declaration adds components to the state, which is the default context for every clause, taking as input/output mode the optional sign in each ~~clause~~[8].

[8] Note also that the example makes use of short syntax for getters and setters

```
:- state calc {
  :- var stack, reg.
  -new :- stack@ = [], reg <- 0.
  +top(X) :- stack@ = [X|_].
  add :- stack@ = [X,Y|L], Z is X+Y, stack <- [Z|L].
  sto :- stack@ = [X|_], reg <- X.
}.
```

Fig. 5. Notation for Implicit Context

3.5 Stateful Logical Loops

Recursion is usually perceived as the main way to express iteration in a declarative setting; however, for some cases, it obfuscates the code and makes it difficult to understand its meaning. Higher-order functions (e.g., `foldr`) emerged as a tool to decompose large programs into smaller and reusable components implementing general data traversal and transformation patterns. When applied to monads, those iteration patterns allow for imperative-like constructions for stateful computations [11].

In the case of Prolog, most modern implementations have some support for higher-order recursion patterns (e.g., `maplist/N`) and some proposals to facilitate loop definitions exist [12], but representing the state is usually left out of the picture. Other approaches, such as monads for λProlog [13], did not materialize as an accepted coding practice for Prolog. Thus, lacking simple mechanisms to manage states and loops, even experienced programmers still rely on dynamic predicates, failure loops, and predicates with dozens of arguments, for tasks that could be described purely using adequate abstraction patterns.

Custom Control Patterns Our approach extends *logical loops* with the possibility for the user to define *control patterns* for stateful computations which involve loops. Their syntax is

```
:- control (H do C) = {
  '' :- ... % default entry point
  <<auxiliary clauses>>
}.
```

where `H` is a term, `C` identifies the goal to iterate, and the definition of the control structure as a set of enclosed clauses (inspired on the module closures described in [14]), with a default entry point identified by the atom ''. The optional arguments in `C` specify existentially quantified variables that are local (existentially quantified) to each call (e.g. loop iteration), and not connected to the rest of the body.

Example 2. What follows is a control pattern which expresses an iteration scheme over lists, as a `for_each(X, List)` that executes a specific goal for each element `X` in the `List` can be defined as:

```
:- control (for_each(X, List) do Code(X)) = {
  '' :- iter(Xs).
  iter([]).
  iter([X|Xs]) :- Code(X), iter(Xs).
}.
```

Each use of the pattern, such as `for_each(X, List) do Goal`, unfolds its definition, creating auxiliary predicates if necessary, containing as parameters and context all the required variables that appear in both *Goal* (visible, that is, not hidden under predicate abstractions or other control patterns) and the rest of the body, except for variable `X`.

Example 3. We write now a simple program `max_prod/3` that obtains the maximum value of the product of elements in two lists, taking advantage of the previous definition of `for_each`. The complexity of the control pattern is hidden when applied to the test program, becomes simpler than its recursive counterpart and reusable with other data structures:

```
max_prod(As, Bs, R) :-
   C <- 0,
   for_each(B, Bs) do for_each(A, As) do max(A * B)/C,
   R = C@ .

max(X)/Z :-
   ( X > Z@ -> Z <- X ; true ).
```

Adopting this syntax can, in some times, render programs which are more readable, shorter, and easy to understand and to explain. As a comparison, the Prolog code corresponding to the previous nested loop predicate is as follows:

```
max_prod_3([], B, C, C).
max_prod_3([E|F], B, C, D) :-
        max(E*B, C, G),
        max_prod_3(F, B, G, D).

max(A, B, C) :-
        ( A > B -> C = A ; C = B ).
```

```
max_prod_3([], B, C, C).
max_prod_3([E|F], B, C, D) :-
        max(E*B, C, G),
        max_prod_3(F, B, G, D).

max(A, B, C) :-
        ( A > B -> C = A ; C = B ).
```

Suppose that at some point it is required to write a version that works with trees (e.g. by generalizing the code for `traverse1` in Fig. 4). While the plain Prolog version of `max_prod/3` is not easy to update, the version using control patterns can be easily modified by declaring a new control pattern:

```
:- control (for_each_node(X, Tree) do Code(X)) = {
  '' :- traverse1 / (+(^Tree)).
  traverse1 / (+nil).
  traverse1 / (+node):-
      traverse1 / (+left),
      get(E) / elem, Code(E),
      traverse1 / (+right).
}.
```

Note that `traverse` predicate can now be written concisely as:

```
traverse / (+tree, +dl) :-
    get(Tree) / +tree,
    for_each_node(E, Tree) do (get([E|L]) / dl, set(L) / dl),
    set([]) / dl.
```

4 Related Work

We will now review and compare with our proposal previously published work which deals explicitly with notions of state and threading.

```
:- acc_info(code, T, Out,
    In, (Out=[T|In])).
:- acc_info(inc_size, T, In,
    Out, (Out is T+In)).
:- pred_info(expr_code, 1,
    [inc_size, code]).

expr_code(A+B) -->>
    expr_code(A),
    expr_code(B),
    [plus]:code,
    [1]:inc_size.
expr_code(I) -->>
    {atomic(I)},
    [push(I)]:code,
    [1]:inc_size.
```

(a) EDCG version

```
:- state code_emit {
  :- var code, size.
  inc_size(N) :- S is size@ + N, size<-S.
  emit(X) :- code@ = [X|Code], code<-Code.
  expr_code(A+B) :-
      expr_code(A),
      expr_code(B),
      emit(plus),
      inc_size(1).
  expr_code(I) :-
      atomic(I),
      emit(push(I)),
      inc_size(1).
}.
```

(b) Version using implicit contexts

Fig. 6. EDCG vs. our approach.

4.1 DCGs and EDCGs

Definite Clause Grammars (DCGs) [15] have been long used, both as a tool for language parsing and as a convenient list-processing technique. DCG clauses represent state with a pair of lists [9] which are implicit predicate arguments and ensure threading of values across literals. The principal operation on the state is removing some head elements from the input list to obtain the output list, which is a very special case of stateful computation compared to the general problem we wish to address in this paper.

Extended DCGs (EDCGs) [2] are a generalization of DCGs which were introduced to enable handling several state components called *accumulators*, represented as input-output pairs of implicit predicate arguments. One operation can be defined for each accumulator. An EDCG clause of the form

```
Head -->> Body.
```

has one pair of input/output arguments for each of the accumulators, where the list of lexical elements is a special case of an accumulator. EDCG clauses use the syntax `[Elem]:Acc` to represent accumulating `Elem` to a particular `Acc`. The accumulators are globally defined with an `acc_info` declaration, and have exactly one accumulation operation. EDCG predicates are declared with a `pred_info` declaration that specifies the accumulators they use. In our approach (see Figure 6 for an example), accumulators in EDCG correspond to contextual state variables, accumulator operations correspond to predicates (which can of course be unfolded) and the `pred_info` corresponds to the declaration of the implicit `emit` context (Section 3.4). EDCGs cannot express directly structured information composed of mutable components, and

[9] In practice, DCGs are often used with all kind of data as a single accumulator, not necessarily lists

the limitation to one operation is too restrictive in a general case where many different operations can be applied to stateful entities, which are, in general, not only accumulators.

4.2 State Variables

State variables, as those in Mercury (see also the global variables of [4]), can represent a mutable state with a pair of logical variables, and thus correspond directly to our notion of a context variable. A state variable in Mercury is a shorthand for naming intermediate values in a sequence, where the *current* and *next* values at a given point are represented as `!.X` and `!:X`, respectively, and `!X` denotes the pair (`!.X`, `!:X`). The compiler transforms the source code to ensure state threading along the same lines as the propagation rules described herein.

However, the most important limitation of state variables in Mercury is that a state variable can appear only as an argument to a predicate, and not as a part of a structured mutable state. For instance, it is not possible to write `!s(X)` to mean the pair (`s(!.X)`, `s(!:X)`). If the state has to be structured, the alternatives are *opening* and deforesting the structure or decomposing and rebuilding it as needed in the program. Both approaches introduce problems akin to those that motivated us to develop the proposed expansion and propagation. Since Mercury is a strongly typed language, a part of the problem can be solved by declaring a type for a state variable that corresponds to a structured term. In Prolog as an untyped language, however, we often need to have clauses that operate on different data structures, possibly defined in different modules. Note that our translation scheme, despite being relatively simple, allows syntactically very nice features such as, e.g., the following variable swap: `^(a,b)<-(b@,a@)`.

4.3 Contextual Logic Programming (CxLP)

Contextual Logic Programming (CxLP) [1] (and the related notion of a *minimal context* [16]) organizes code in parametrized units and resolves predicate calls in a current execution context to a particular predicate defined in a particular unit. CxLP is a major alternative to the existing, more traditional, Prolog execution engines and module systems, and is able to represent complex operations on state encoded in unit parameters. However, in our approach, we wish to be as transparent as possible with respect to the underlying Prolog system and to avoid reliance on specialized abstract machine instructions and global variables. Besides, CxLP is generally not concerned with modeling mutable state, threading, and structured state representations. Note that, as reported by their authors, the strength of the dynamically configurable structure of CxLP is also its weakness, since that makes techniques such as static analysis (and even separate compilation) very difficult.

5 Conclusions

We have presented an approach for easily expressing the notion of complex state change within a Logic Programming language. This proposal has been used to (re)implement non-trivial software systems, such as the LPDoc autodocumenter [8] or the ImProlog compiler [17]. Both belong to the large and relevant class of systems

which perform extensively transformations on states, and both have reportedly benefited from a cleaner way of specifying this state. We have also given examples of how well-known programming constructs which are also relevant for state-transforming algorithms, such as loops and (E)DCGs, can be cast into our approach. The implementation of such an approach turns out to be quite simple. In our case we have resorted to using a compile-time program transformation which works at the module level using the facilities of the Ciao system.

References

1. Monteiro, L., Porto, A.: Contextual logic programming. In Levi, G., Martelli, M., eds.: Logic Programming: Proc. of the Sixth International Conference. MIT Press, Cambridge, MA (1989) pp. 284–299
2. Van Roy, P.: A prolog compiler for the plm. Report no. ucb/csd 84/203, Univ. of California, Berkeley (1984)
3. Henderson et al., F.: The Mercury Project Documentation. URL: `http://www.cs.mu.oz.au/research/mercury/information/documentation.html`.
4. Schachte, P.: Global Variables in Logic Programming. In: ICLP'97, pp. 3–17. MIT Press (1997)
5. Aggoun, A., Beldiceanu, N.: Time stamps techniques for the trailed data in constraint logic programming systems. In Bourgault, S., Dincbas, M., eds.: SPLT. pp. 487–510. (1990)
6. Hermenegildo, M., Puebla, G., Bueno, F., López-García, P.: Program Development Using Abstract Interpretation (and The Ciao System Preprocessor). In: Proc. of SAS'03, pp. 127–152. Springer LNCS 2694 (2003)
7. Mera, E., López-García, P., Hermenegildo, M.: Towards Integrating Run-Time Checking and Software Testing in a Verification Framework. Technical Report CLIP1/2009.0, Technical University of Madrid (UPM), School of Computer Science, UPM (March 2009)
8. Hermenegildo, M.: A Documentation Generator for (C)LP Systems. In: International Conference on Computational Logic, CL2000. Number 1861 in LNAI, pp. 1345–1361. Springer-Verlag (July 2000)
9. Vaucheret, C., Bueno, F.: More Precise yet Efficient Type Inference for Logic Programs. In: Proc. of SAS'02, pp. 102–116. Springer LNCS 2477 (2002)
10. Mera, E., López-García, P., Hermenegildo, M.: Integrating Software Testing and Run-Time Checking in an Assertion Verification Framework. In: International Conference on Logic Programming (ICLP). LNCS. Springer-Verlag (July 2009) To appear.
11. Wadler, P.: How to declare an imperative. ACM Comput. Surv. **29**(3), pp. 240–263 (1997)
12. Schimpf, J.: Logical loops. In: In Proceedings of the 18th International Conference on Logic Programming, ICLP 2002, pp. 224–238. SpringerVerlag (2002)
13. Bekkers, Y., Tarau, P.: Monadic constructs for logic programming. In Lloyd, J., ed.: Proceedings of ILPS'95, pp. 51–65. MIT Press (December 1995)
14. Haemmerlé, R., Fages, F., Soliman, S.: Closures and modules within linear logic concurrent constraint programming. In: FSTTCS. pp. 544–556. (2007)
15. Abramson, H.: Definite clause translation grammars. In: International Symposium on Logic Programming, Silver Spring, MD, pp. 233–242. IEEE Computer Society (February 1984)
16. Abreu, S., Diaz, D.: Objective: in minimum context. In: In: Proc. Nineteenth International Conference on Logic Programming, pp. 128–147. MIT Press (2003)
17. Morales, J., Carro, M., Hermenegildo, M.: Towards Description and Optimization of Abstract Machines in an Extension of Prolog. In Puebla, G., ed.: Logic-Based Program Synthesis and Transformation (LOPSTR'06). Number 4407 in LNCS pp. 77–93. (July 2007)